Problem Solving

Announcements

Tree Processing

Using the Tree Abstraction

For a tree t, you can **only**:

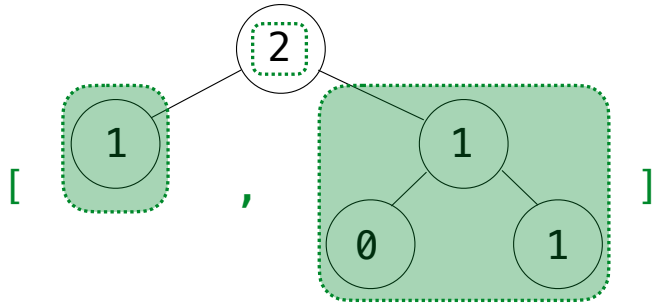
- Get the label for the root of the tree: `t.label`
- Get the list of branches for the tree: `t.branches`
- Get the branch at index i, which is a Tree: `t.branches[i]`
- Determine whether the tree is a leaf: `is_leaf(t)`
- Treat t as a value: `return t`, `f(t)`, `[t]`, `s = t`, etc.

Create a tree by calling Tree on a label and a list of branches

```
t = Tree(2, [Tree(1), Tree(1, [Tree(0), Tree(1)])])
```

isinstance(t, Tree) evaluates to True

An example tree `t`:

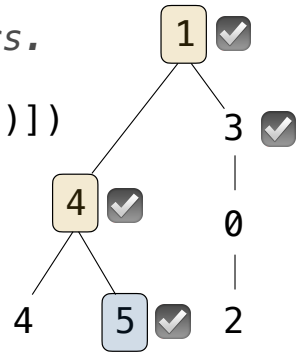


Solving Tree Problems

Implement **bigs**, which takes a Tree instance t containing integer labels. It returns the number of nodes in t whose labels are larger than all labels of their ancestor nodes. (Assume the root label is always larger than all its ancestors, since it has no ancestors.)

```
def bigs(t: Tree[int]):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
```




```
    if is_leaf(t):
        return ___
    else:
        return ___([___ for b in t.branches])
```

Somehow increment the total count

Somehow track a list of ancestors

```
if node.label > max(ancestors):
```

Somehow track the largest ancestor

```
if node.label > max_ancestor:
```

Solving Tree Problems

Implement **bigs**, which takes a Tree instance t containing integer labels. It returns the number of nodes in t whose labels are larger than all labels of their ancestor nodes. (Assume the root label is always larger than all its ancestors, since it has no ancestors.)

```
def bigs(t: Tree[int]):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):

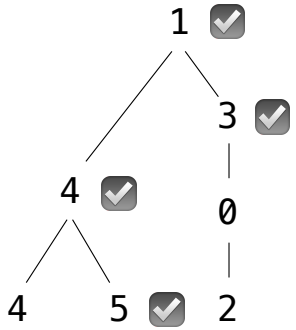
        if ____________________________________________________:

            return 1 + ____________________________________________

        else:

            return ____________________________________________

    return ____________________________________________________
```



Solving Tree Problems

Implement **bigs**, which takes a Tree instance t containing integer labels. It returns the number of nodes in t whose labels are larger than all labels of their ancestor nodes. (Assume the root label is always larger than all its ancestors, since it has no ancestors.)

```
def bigs(t: Tree[int]):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        if ____________________:
            return 1 + ____________________
        else:
            return ____________________
    return ____________________
```

Somehow track the largest ancestor

A node

max_ancestor

a.label > x

node.label > max_ancestor

sum([f(b, a.label) for b in a.branches])

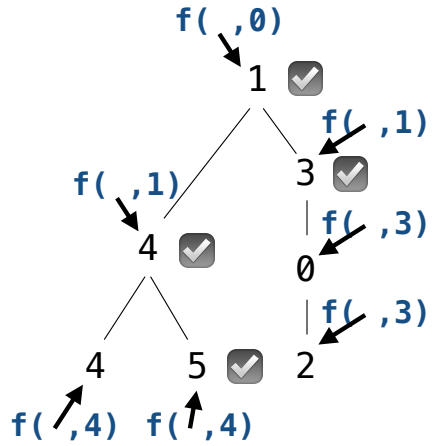
Somehow increment the total count

sum([f(b, x) for b in a.branches])

f(t, t.label - 1)

Root label is always larger than its ancestors

Some initial value for the largest ancestor so far...



How to Design Programs

Problem Solving Process from *How to Design Programs*

From Problem Analysis to Data Definitions
Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

Signature, Purpose Statement, Header
State what kind of data the desired function consumes and produces. Formulate a concise answer to the question *what* the function computes. Define a stub that lives up to the signature.

Functional Examples
Work through examples that illustrate the function's purpose.

Function Template
Translate the data definitions into an outline of the function.

Function Definition
Fill in the gaps in the function template. Exploit the purpose statement and the examples.

Testing
Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

https://htdp.org/2026-5-28//Book/part_preface.html#%28part._sec~3asystematic-design%29

Applying the Design Process

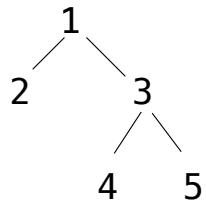
61A Fall 2015 Final Question 3 [Extended Remix]

Definition. A *path* through a Tree is a list of adjacent node labels that starts with the root label and ends with a leaf label.

```
def count_big(t, n):  Signature: (Tree, minimum path sum) -> number of paths
    """Return the number of paths in t that have a sum larger or equal to n.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> count_big(t, 3)
    3
    >>> count_big(t, 6)
    2
    >>> count_big(t, 9)
    1
    """
    if is_leaf(t):
        return one(_____________________________________________________________)
    else:
        return _____________________________________________________________
```

```
def one(b):
    if b:
        return 1
    else:
        return 0
```



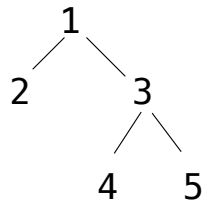
61A Fall 2015 Final Question 3 [Extended Remix]

Definition. A *path* through a Tree is a list of adjacent node labels that starts with the root label and ends with a leaf label.

```
def count_big(t, n):  Signature: (Tree, minimum path sum) -> number of paths
    """Return the number of paths in t that have a sum larger or equal to n.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> count_big(t, 3)
    3
    >>> count_big(t, 6)
    2
    >>> count_big(t, 9)
    1
    """
    if is_leaf(t):
        return one(_____t.label >= n_____)
    else:
        return ____sum([count_big(b, n-t.label) for b in t.branches])____
```

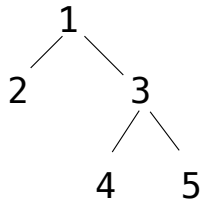
```
def one(b):
    if b:
        return 1
    else:
        return 0
```



61A Fall 2015 Final Question 3 [Extended Remix]

```
def print_big(t, n):
    """Print the paths in t that have a sum larger or equal to n.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> print_big(t, 3)
    [1, 2]
    [1, 3, 4]
    [1, 3, 5]
    >>> print_big(t, 6)
    [1, 3, 4]
    [1, 3, 5]
    >>> print_big(t, 9)
    [1, 3, 5]
    """
    def helper(t, p):
        ____________________
        if is_leaf(t):
            if _______________:
                print(p)
        else:
            ____________________:
                ____________________________________________________
    helper(t, ____)
```



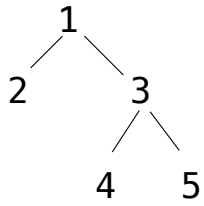
Signature: (Tree, list[int]) -> None

"For a node & partial path, extend the path & maybe print it"

61A Fall 2015 Final Question 3 [Extended Remix]

```
def print_big(t, n):
    """Print the paths in t that have a sum larger or equal to n.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> print_big(t, 3)
    [1, 2]
    [1, 3, 4]
    [1, 3, 5]
    >>> print_big(t, 6)
    [1, 3, 4]
    [1, 3, 5]
    >>> print_big(t, 9)
    [1, 3, 5]
    """
    def helper(t, p):
        p = p + [t.label]
        if is_leaf(t):
            if sum(p) >= n:
                print(p)
        else:
            for b in t.branches:
                helper(b, p)
    helper(t, [])
```



Signature: (Tree, list[int]) -> None

"For a node & partial path, extend the path & maybe print it"

Recursion Practice

Spring 2023 Midterm 2 Question 5(a) [modified a bit]

Definition. When parking vehicles in a row, a motorcycle takes up 1 parking spot and a car takes up 2 adjacent parking spots. A string of length n can represent n adjacent parking spots using % for a motorcycle, <> for a car, and . for an empty spot.

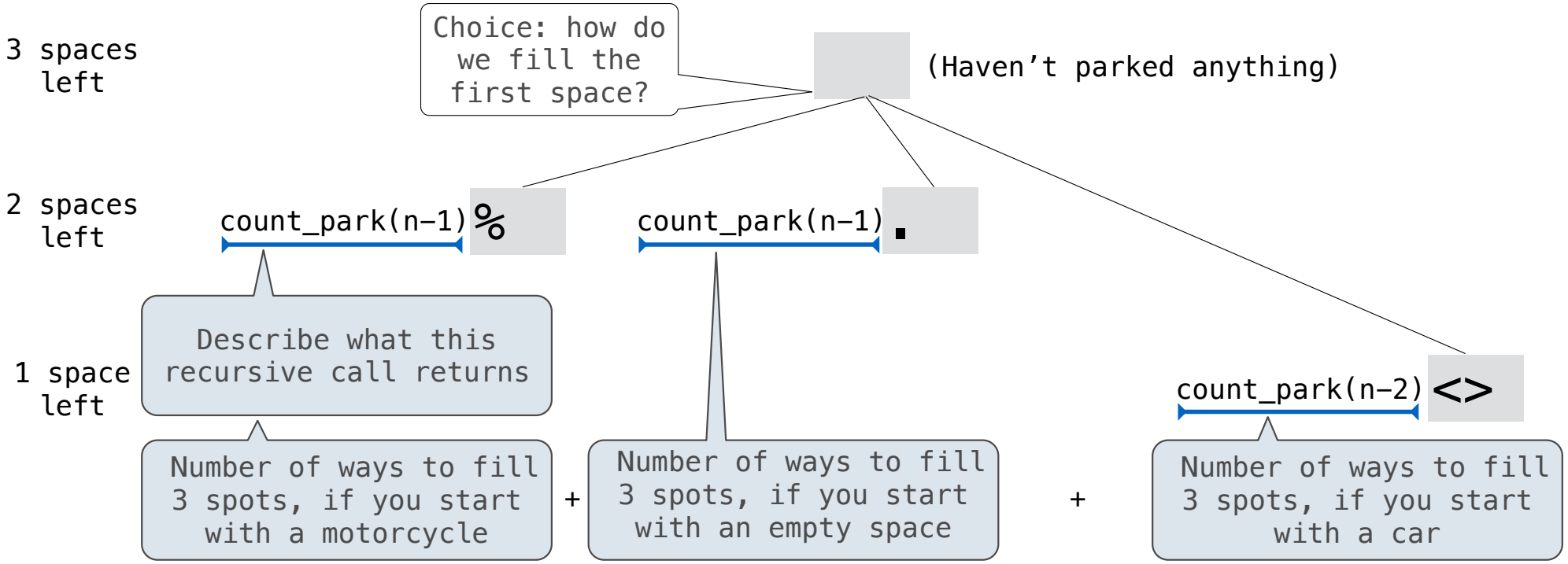
For example: '.%%.<><>' (Thanks to the Berkeley Math Circle for introducing this question.)

Implement **count_park**, which returns the number of ways that vehicles can be parked in n adjacent parking spots for positive integer n. Some or all spots can be empty.

```
def count_park(n):
    """Count the ways to park cars and motorcycles in n adjacent spots.
    >>> count_park(1)  # '.' or '%'
    2
    >>> count_park(2)  # '..', '.%', '%.', '%%', or '<>'
    5
    >>> count_park(4)  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """
```

We haven't parked anything yet. What's a first choice we can make?

Spring 2023 Midterm 2 Question 5(a) [modified a bit]



Spring 2023 Midterm 2 Question 5(a) [modified a bit]

Definition. When parking vehicles in a row, a motorcycle takes up 1 parking spot and a car takes up 2 adjacent parking spots. A string of length n can represent n adjacent parking spots using % for a motorcycle, <> for a car, and . for an empty spot.

For example: '.%%.<><>' (Thanks to the Berkeley Math Circle for introducing this question.)

Implement **count_park**, which returns the number of ways that vehicles can be parked in n adjacent parking spots for positive integer n. Some or all spots can be empty.

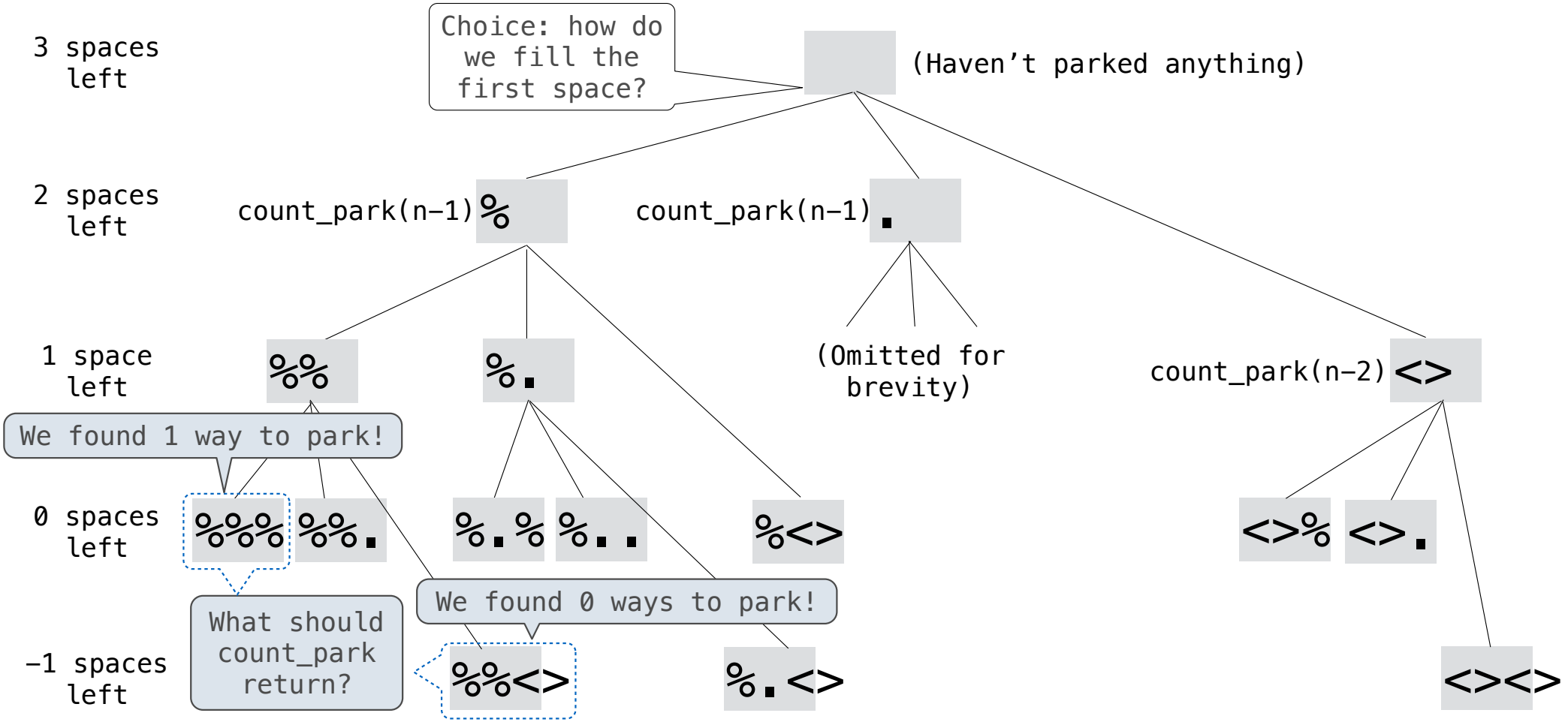
```
def count_park(n):
    """Count the ways to park cars and motorcycles in n adjacent spots.
    >>> count_park(1)  # '.' or '%'
    2
    >>> count_park(2)  # '..', '.%', '%.', '%%', or '<>'
    5
    >>> count_park(4)  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """
    if n < 0:
        return _____________
    elif n == 0:
        return _____________
    else:
        return count_park(n-1) + count_park(n-1) + count_park(n-2)
```

One way to think about these base cases: lead to these cases, their values be?

Number of ways to fill 3 spots, if you start with a motorcycle

```
count_park(3):
  %%%
  %%.
  %.%
  %..
  %<>
  .%%
  .%.
  ..%
  ...
  .<>
  <>%
  <>.
```

Spring 2023 Midterm 2 Question 5(a) [modified a bit]



Spring 2023 Midterm 2 Question 5(a) [modified a bit]

Definition. When parking vehicles in a row, a motorcycle takes up 1 parking spot and a car takes up 2 adjacent parking spots. A string of length n can represent n adjacent parking spots using % for a motorcycle, <> for a car, and . for an empty spot.

For example: '.%%.<><>' (Thanks to the Berkeley Math Circle for introducing this question.)

Implement **count_park**, which returns the number of ways that vehicles can be parked in n adjacent parking spots for positive integer n. Some or all spots can be empty.

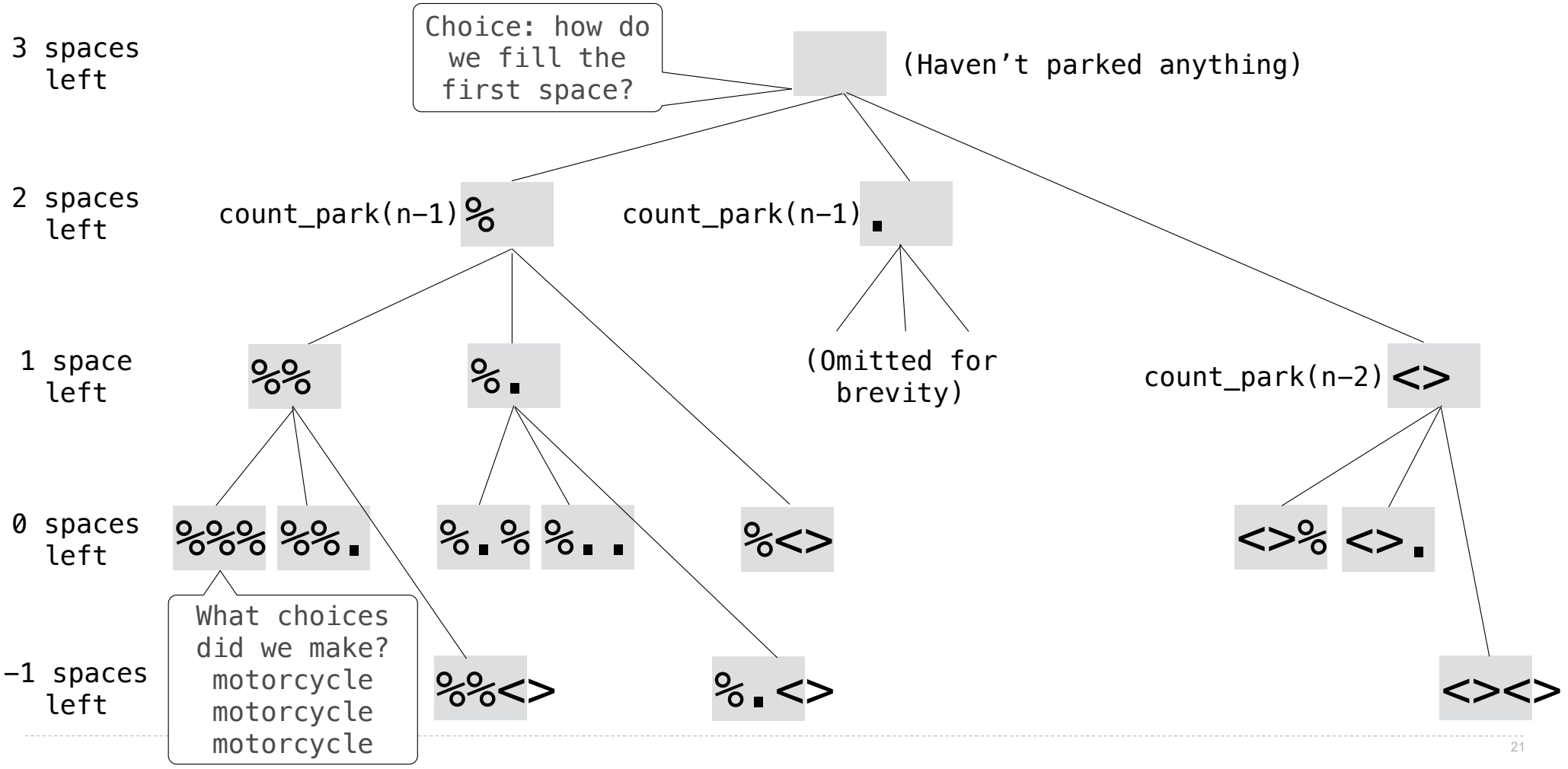
```
def count_park(n):
    """Count the ways to park cars and motorcycles in n adjacent spots.
    >>> count_park(1)  # '.' or '%'
    2
    >>> count_park(2)  # '..', '.%', '%.', '%%', or '<>'
    5
    >>> count_park(4)  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """
    if n < 0:
        return ____0____
    elif n == 0:
        return ____1____
    else:
        return count_park(n-1) + count_park(n-1) + count_park(n-2)
```

One way to think about these base cases: which recursive calls lead to these cases, and what should their values be?

count_park(3):

%%%
%%.
%.%
%..
%<>
.%%
.%.
..%
...
.<>
<>%
<>.

Spring 2023 Midterm 2 Question 5(a) [modified a bit]



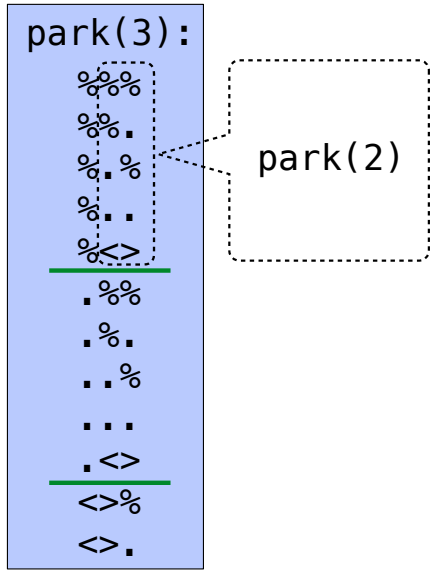
Spring 2023 Midterm 2 Question 5(b) [modified a lot]

Definition. When parking vehicles in a row, a motorcycle takes up 1 parking spot and a car takes up 2 adjacent parking spots. A string of length n can represent n adjacent parking spots using % for a motorcycle, <> for a car, and . for an empty spot.

For example: '.%%.<><>' (Thanks to the Berkeley Math Circle for introducing this question.)

Implement **park**, which returns a list of all the ways, represented as strings, that vehicles can be parked in n adjacent parking spots for positive integer n. Spots can be empty.

```
def park(n):
    """Return the ways to park cars and motorcycles in n adjacent spots.
    >>> park(1)
    ['%', '.']
    >>> park(2)
    ['%%', '%.', '.%', '..', '<>']
    >>> len(park(4))  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """
    if n < 0:
        return _______
    elif n == 0:
        return _______
    else:
        return ____________________________________________________________________
```



motorcycle first + nothing first + car first

Spring 2023 Midterm 2 Question 5(b) [modified a lot]

Definition. When parking vehicles in a row, a motorcycle takes up 1 parking spot and a car takes up 2 adjacent parking spots. A string of length n can represent n adjacent parking spots using % for a motorcycle, <> for a car, and . for an empty spot.

For example: '.%%.<><>' (Thanks to the Berkeley Math Circle for introducing this question.)

Implement **park**, which returns a list of all the ways, represented as strings, that vehicles can be parked in n adjacent parking spots for positive integer n. Spots can be empty.

```
def park(n):
    """Return the ways to park cars and motorcycles in n adjacent spots.
    >>> park(1)
    ['%', '.']
    >>> park(2)
    ['%%', '%.', '.%', '..', '<>']
    >>> len(park(4))  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """
    if n < 0:
        return []
    elif n == 0:
        return ['']
    else:
        return ['%'+s for s in park(n-1)] + ['.'+s for s in park(n-1)] + ['<>'+s for s in park(n-2)]
```

motorcycle first + nothing first + car first

